KICK ASS JAVA PROGRAMMING

CD-ROM INCLUDED

Advanced everything—with an attitude!

Tonny Espeset

CORIOLIS GROUP BOOKS

CHAPTER 7

Adding Realism

CHAPTER 7

- **Using normal vectors**
- **Calculating surface illumination**
- **Adding shadows**
- **3D modeling**
- **Texture mapping**

Adding Realism

If you're familiar with VRML format and VRML viewers, you know that the best viewers can cram a lot of realism into any 3D object by using advanced rendering techniques. Compared to Java, these viewers have one huge advantage: Because they're partly written in a low-level language, the rendering speed is improved dramatically. You can make a 3D viewer just as advanced—or even more advanced—using Java, but the frame rate would soon drop to a level where you'd more likely be talking about seconds per frame rather than frames per second. The trick is to make the VRML engine just a tad less general. For instance, the renderer presented in this chapter will not deal with objects passing through other objects. This is a rather small price to pay for real-time performance, and you can create all your objects without depending on this feature.

For ultimate realism, you must set your 3D scenes in an environment with light, shadows, and solid surfaces. That's not easy in any language, and Java users have struggled to keep rendering speed up. Now, the struggle is over.

Turn On the Light

You've already learned all the basics of how to construct an object from points forming surfaces. To render a solid model, simply paint each polygon using **.fillPolygon** instead of using **.drawPolygon** as you learned ear-

lier. But filling all polygons with the same color would just create a silhouette of the object. You can't depend on the VRML object to contain color information, either. Even if it did, not all polygons would differ in color anyway, making them impossible to tell apart.

Normally, not all the surfaces in the object are parallel to each other, which means they all face the camera at different angles. What if we changed the intensity of each surface according to this angle? An angle of 90 degrees could produce a bright color, and if the surface faces the other way (+/-180 degrees), the resulting color should be at its darkest. This would give most faces a unique color, enabling the user to distinguish between them and better see the details in the scene. What we just did, in essence, was to place a light source at the position of the camera. You could move it to any other point in the scene to change the angle of the light. The light source is a point in 3D space just like any other: You can move the light source around to create interesting effects.

The only thing we need to figure out is how to find the angle between the incoming light and each surface. Once we know this, it's very easy to find the correct color intensity. An angle of 90 degrees will produce a brightness factor of 1.0, an angle of 180 degrees will make the brightness factor 0.0, and an angle of 270 degrees will give the brightness factor a value of 1.0. If you've played around with sinus (the trigonometric "sine" function, for all you parochial types), you probably know that the brightness factor can be calculated with just one line of code:

```
double brightness=Math.sin(angle);
```

Once you know the brightness factor, you can use it to multiply the red, green, and blue color components of the surface. Negative color components are then replaced with zeros, and the correct color of the surface relative to its light source will result. If you have multiple light sources in the scene, this process should be repeated, with the final color being the sum of all the calculated color components. (Of course, each red, green, and blue component must be added independently.)

The problem now is to find the correct angle between the light and the surface. This is a somewhat more complicated process, and to understand it you need to know a bit more about a surface's *normal vector*

Just a Normal Vector

Obviously, you need to know which way a surface is facing to know how much it is illuminated by a point light source. Just think of a simple cube with one of its surfaces facing you, then imagine a light source placed above your head. Of course, the surface you're watching should be clear and bright, because nothing is placed between that surface and the light. Now imagine that the light remains stationary while you walk around to the other side of the cube. When you look at it from this side, it will be completely black because no light passes through the cube from the other side. This is all logical for us, but how about for the computer? Let's take a look at how such a scene would be represented in our 3D format. As you know, each surface in the cube is formed by four points. Now consider the two surfaces you just observed: the front surface that faces the light, and the back surface that faces the away from it. The problem is that they're both constructed the exact same way! So how can we tell them apart? The answer is, we can't—at least, not yet. First we'll need to add some more information to each surface, namely a normal vector.

Vectors

The three components of a vector define its magnitude and direction, and any point in space can be described by a vector. Its magnitude is the point's distance from the center point (0, 0, 0), and its direction describes its position relative to the same center point.

Think of a normal vector as an arrow pointing from a surface to show which side is its front. This arrow is perpendicular to the surface. It will be constructed just like any other object in the scene, and because it's just a straight line, two points are sufficient to describe it. The first point is placed in the middle of the surface. You can find this middle point by summing up all x, y, and z coordinates belonging to the vertices forming the surface. Each of these sums is then divided by the number of vertices, and the results form the surface's middle point coordinate. So if a surface is described by the points (x1, y1, z1), (x2, y2, z2), (x3, y3, z3), and (x4, y4, z4), the middle point (x, y, z) is found like this:

```
x=(x1+x2+x3+x4)/4;
y=(y1+y2+y3+y4)/4;
z=(z1+z2+z3+z4)/4;
```

Once you find the middle point of the surface, you have to figure out the coordinate of the tip of the arrow that points out of the surface. In mathematical terms, you need to find the normal vector given at least three vectors describing the plane of the surface. If three vertices of the surface are given by the position vectors **v1**, **v2**, and **v3**, then you can calculate the normal vector **n** by the cross product:

```
n = (v1 - v2) x (v1 - v3)
```

If all this talk about vectors and cross products makes no sense to you, don't worry. It's just a convenient way of describing how to calculate the normal vector. When the **model** array contains all vertices' coordinates and the **p** array contains the indexes to the points forming the polygon, the normal vector [nx, ny, nz] can be found using this code snippet:

```
double vectorX1=model[p[0]*3];
double vectorY1=model[p[0]*3+1];
double vectorZ1=model[p[0]*3+2];

double vectorX2=model[p[1]*3];
double vectorY2=model[p[1]*3+1];
double vectorZ2=model[p[1]*3+2];

double vectorX3=model[p[2]*3];
double vectorY3=model[p[2]*3+1];
double vectorZ3=model[p[2]*3+2];

double px=vectorX1-vectorX2;
double py=vectorY1-vectorY2;
double pz=vectorZ1-vectorZ2;

double qx=vectorX1-vectorX3;
double qy=vectorY1-vectorY3;
double qz=vectorZ1-vectorZ3;

double nx=py*qz-pz*qy;
double ny=pz*qx-px*qz;
double nz=px*qy-py*qx;
```

That code equals the formula you were shown earlier:

```
n = (v1 - v2) x (v1 - v3)
```

and proves that mathematical notation can come in handy to explain things more briefly.

We've now found the location of the normal vector's tip, but how far out from the surface is it sticking out? Its magnitude can be found using this line of code:

```
double length=Math.sqrt(nx*nx+ny*ny+nz*nz);
```

Now you can make sure the magnitude of the normal vector is always one. By setting the length of all normal vectors in the object to this same value, things will be less complicated later on:

```
double scale=1.0/length;
nx*=scale;
ny*=scale;
nz*=scale;
```

The line pointing out from the surface is now given by the surface's midpoint (x, y, z) and the point (nx+x, ny+y, nz+z). But how do you know which side of the surface is considered out? The answer is: The way the points are connected in the surface. To visualize this, take a look at your left hand. Point your thumb straight up while slightly bending your fingers. Your thumb represents the normal vector of the surface, and if you let your fingers rotate along with the points making up the surface, you'll see which way the normal vector will point.

In other words, your object's surface must be constructed using polygons defined in a clockwise or counterclockwise direction. Let's review an example with a simple cube.

The normal vectors belonging to a cube will all point outward from its center, as shown in Figure 7.1.

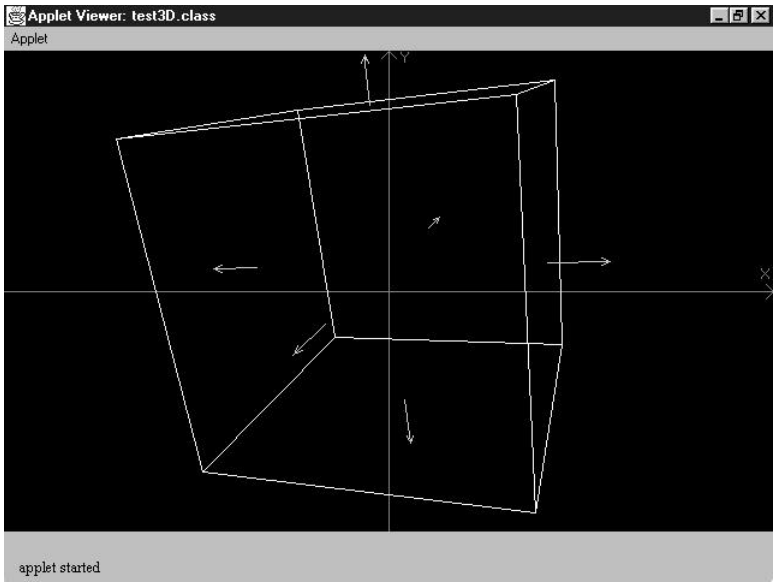

FIGURE 7.1

A cube with normal vectors attached.

You can create a method for adding a cube to a scene that will also do all the required initializations of the surface normal vectors. This method might come in handy later, too, if you want to construct a scene of cubes without having to load a VRML file first. If the center point of the cube is given by (**cX**, **cY**, **cZ**) and its size is given by **size**, the code for adding the cube to the scene will look like Listing 7.1.

Listing 7.1 Creating a cube.

```
int[] polygon=new int[4];
int vertOffset=vertices;

//Set up vertices to form cube
addVertex(cX-size,cY-size,cZ-size);
addVertex(cX-size,cY+size,cZ-size);
addVertex(cX+size,cY+size,cZ-size);
addVertex(cX+size,cY-size,cZ-size);

addVertex(cX-size,cY-size,cZ+size);
addVertex(cX-size,cY+size,cZ+size);
addVertex(cX+size,cY+size,cZ+size);
```

```
addVertex(cX+size,cY-size,cZ+size);

//Set up polygons to form cube
polygon[0]=vertOffset+0;
polygon[1]=vertOffset+1;
polygon[2]=vertOffset+2;
polygon[3]=vertOffset+3;
addPolygon(polygon);

polygon[0]=vertOffset+7;
polygon[1]=vertOffset+6;
polygon[2]=vertOffset+5;
polygon[3]=vertOffset+4;
addPolygon(polygon);

polygon[0]=vertOffset+4;
polygon[1]=vertOffset+5;
polygon[2]=vertOffset+1;
polygon[3]=vertOffset+0;
addPolygon(polygon);

polygon[0]=vertOffset+2;
polygon[1]=vertOffset+6;
polygon[2]=vertOffset+7;
polygon[3]=vertOffset+3;
addPolygon(polygon);

polygon[0]=vertOffset+1;
polygon[1]=vertOffset+5;
polygon[2]=vertOffset+6;
polygon[3]=vertOffset+2;
addPolygon(polygon);

polygon[0]=vertOffset+3;
polygon[1]=vertOffset+7;
polygon[2]=vertOffset+4;
polygon[3]=vertOffset+0;
addPolygon(polygon);
```

The method setting up the cube will store its vertices in the model array in the sequence indicated in Figure 7.2.

When the cube is set up with the vertices as indicated in Figure 7.2, you need to specify which vertices are to form which surfaces. As you can see,

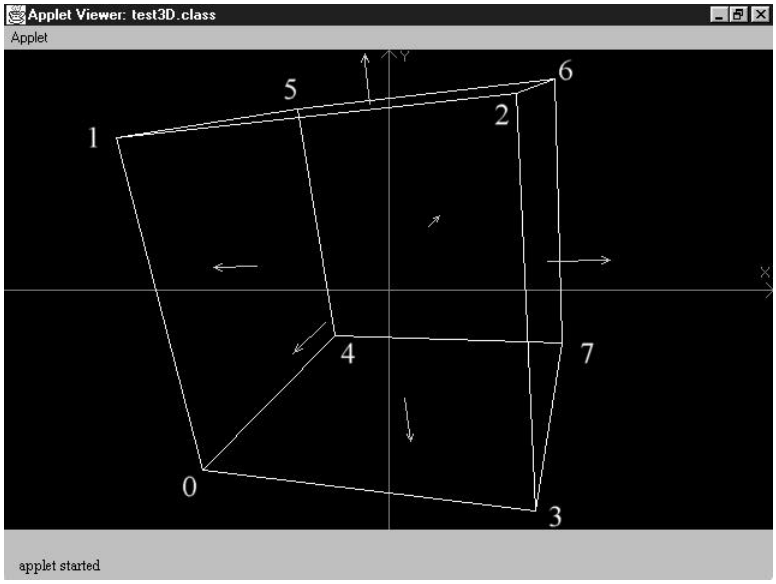

FIGURE 7.2

The cube vertices.

the front surface is formed by the vertices with offsets 0, 1, 2, and 3. Note that the order is significant for which way the normal vector is to point. By defining the surface in a clockwise order like this, you'll have the normal vector face the right way, toward the viewer (remember the left thumb rule). The back of the cube will be defined by the points 7, 6, 5, and 4. Notice how the direction in the point sequence forming the surface changed. Apply the left thumb rule again, and you'll see that the normal vector now points away from the viewer. You can check all the surfaces in the scene using the left thumb rule to verify that all normal vectors point away from the center of the cube when defined this way. Call it a "rule of thumb."

The **.addPolygon** method simply calculates the normal vector from the given polygon and then adds it to the **normal** array. This array is constructed the same way as the **model** array, except each normal vector is described using two points instead of one, and therefore uses six array elements each. The first three elements describe the coordinate to the surface's midpoint, the next three describe the location of the normal vector's tip. The complete **.addPolygon** method is shown in Listing 7.2.

Listing 7.2 Adding a polygon.

```
void addPolygon(int[] p)
{
    //The polygon is added and a normal vector is constructed to
    //point according to the left thumb rule
    int nVertices=p.length;
    double midX=0,midY=0,midZ=0;
    for (int n=0;n<nVertices;n++)
    {
        addPolyVert(p[n]);
        midX+=model[p[n]*3];
        midY+=model[p[n]*3+1];
        midZ+=model[p[n]*3+2];
    }
    addPolyVert(-1);

    //Find the midpoint of the polygon
    midX/=nVertices;
    midY/=nVertices;
    midZ/=nVertices;

    //Find normal vector
    double vectorX1=model[p[0]*3];
    double vectorY1=model[p[0]*3+1];
    double vectorZ1=model[p[0]*3+2];

    double vectorX2=model[p[1]*3];
    double vectorY2=model[p[1]*3+1];
    double vectorZ2=model[p[1]*3+2];

    double vectorX3=model[p[2]*3];
    double vectorY3=model[p[2]*3+1];
    double vectorZ3=model[p[2]*3+2];

    double px=vectorX1-vectorX2;
    double py=vectorY1-vectorY2;
    double pz=vectorZ1-vectorZ2;

    double qx=vectorX1-vectorX3;
    double qy=vectorY1-vectorY3;
    double qz=vectorZ1-vectorZ3;

    double nx=py*qz-pz*qy;
    double ny=pz*qx-px*qz;
    double nz=px*qy-py*qx;
```

```
    double length=Math.sqrt(nx*nx+ny*ny+nz*nz);
    double scale=1.0/length;

    nx*=scale;
    ny*=scale;
    nz*=scale;

    addNormal(midX,midY,midZ,midX+nx,midY+ny,midZ+nz);

    polygons++;
}
```

The **.addNormal** method called from **.addPolygon** (shown in Listing 7.3) pretty much does the same thing as the **.addVertex** method listed earlier. It simply adds the points forming the normal vector to the **normal** array, and expands this array if necessary.

Listing 7.3 Adding a normal.

```
void addNormal(double x1, double y1, double z1,double x2,
          double y2, double z2)
{
     if (normals<maxNormals)
     {
          normal[normals*6]=x1;
          normal[normals*6+1]=y1;
          normal[normals*6+2]=z1;

          normal[normals*6+3]=x2;
          normal[normals*6+4]=y2;
          normal[normals*6+5]=z2;
     }
     else
     {
          maxNormals+=500;
          double[] normalTemp=new double[maxNormals*6];
          System.arraycopy(normal,0,normalTemp,0,normals*6);

          normalTemp[normals*6]=x1;
          normalTemp[normals*6+1]=y1;
          normalTemp[normals*6+2]=z1;

          normalTemp[normals*6+3]=x2;
          normalTemp[normals*6+4]=y2;
          normalTemp[normals*6+5]=z2;
```

```
            normal=normalTemp;
            transNormal=new double[maxNormals*6];
        }
        normals++;
}
```

Note that you must transform all the points in the **normal** vector array, along with the points forming the scene. The transformed normal vectors are then placed in the **transNorm** array.

Calculating Surface Illumination

Now, you can create a cube ready to be illuminated by a light source. When a surface in the cube is to be painted, you just have to calculate the **illumination** factor based on the angle between the normal vector and the incoming light ray. You can think of the light ray as a line passing through both the light source and the surface's midpoint. The **illumination** factor is the cosine of the angle **a** between this light ray and the surface's normal vector. If **n** is the normal vector and **l** is the vector describing the direction of the light, then the cosine of the angle between them is given by the equation:

```
n ∑ l = |n| |l| cos(a)
```

If you're unfamiliar with vector arithmetic's, n • l means the dot product of the vector **n** and **l**. If **n** is defined by [nx, ny, nz] and **l** is defined by [lx, ly, lz], then the dot product of **n** and **l** is given by nx*lx+ny*ly+nz*lz. The term |n| and |l| means the magnitude of **n** and **l** respectively. The magnitude of **l** can be found with magnitude=Math.sqrt(lx*lx+ly*ly+lz*lz). As you can see, finding the magnitude of a vector forces you to use a square-root calculation. This is a slow process for the computer, but luckily we know that the magnitude of the normal vector is always 1, which helps to simplify the calculation. The cosine of the angle **a** can now be found using this formula:

```
 n ∑ l
cos(a) =     -------
 |l|
```

In Java code, this calculation looks somewhat more complicated. If the transformed normal vectors are placed in the **transNorm** array and **offset** points to the surface's normal vector, then the **illumination** factor to this surface

with respect to the light source located at (**lightX**, **lightY**, **lightZ**) is found with this code snippet:

```
double normalx=transNormal[offset+3]-transNormal[offset];
double normaly=transNormal[offset+3+1]-transNormal[offset+1];
double normalz=transNormal[offset+3+2]-transNormal[offset+2];

dx=lightX-transNormal[offset];
dy=lightY-transNormal[offset+1];
dz=lightZ-transNormal[offset+2];
double illumination=(dx*normalx+dy*normaly+dz*normalz)/
        Math.sqrt(dx*dx+dy*dy+dz*dz);
```

The **illumination** variable will now hold a value ranging from -1.0 to 1.0, describing how much light was received from the light source at the midpoint. The surface can now be drawn in the appropriate color grade.

By adding some shading, as shown in Figure 7.3, the cube looks a lot more realistic than the wire-frame version.

Actually, this is not an entirely accurate way of illuminating an object. To do it properly, you need to calculate each pixel in the plotted surface independently. For example, each of the four corners in one of the surfaces

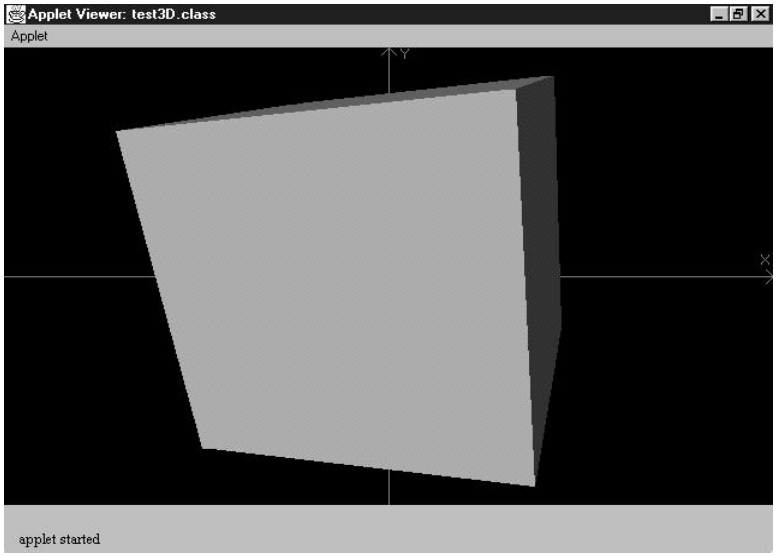

FIGURE 7.3

A flat-shaded cube.

forming the cube would have its own angle to the incoming light, and would, therefore, not be illuminated equally. Unfortunately, calculating the amount of light striking each pixel in all surfaces would cause the computer to choke on the numbers. Even when written in machine code, such accurate rendering needs heavy optimizing for anything close to real-time performance. In most cases, using just the surface's midpoint to calculate the amount of light striking it is more than adequate. We'll stick to this technique, even though it means we'll have no spectacular reflections. If you really need spectacular reflections anyway, you can increase the amount of surfaces in your object. The more surfaces, the more accurate the shading of the model will be. But again, the more surfaces you add, the slower the rendering process will be.

Illuminating VRML Objects

While cubes are a nice start, you'll no doubt want to be able to display more complex objects. So how can we improve the VRML loader to initialize the object for a scene with light sources? As I mentioned earlier, all you need to do is find the right normal vector for each surface in the scene. This could have caused some major headaches if it weren't for the intelligent way VRML objects are stored. Each surface is formed by points in a clockwise or counterclockwise manner, depending on which side faces out, just as in the cube we constructed earlier. This makes finding all the normal vectors very simple. In fact, you can use the exact same calculations as performed by **.addPolygon**.

So far, we've used a left-thumb rule to figure out which way the normal vector points, but some VRML scenes are inverted, so a right-thumb rule must be used. If you use the right-thumb rule, you need to turn all normal vectors in the opposite direction. To do so, you simply multiply the normal vector by -1. But how do you know which rule to use? Generally, you don't. It's easiest to just change the normal vector sign when needed. If you stick to one VRML converter or saver, you'll only have to make sure once that the sign of the normal vector is compatible with its way of storing the object. So if the scene looks "inside out," changing the sign of the normal vectors might be a good idea. This is easy, since the **.createNormals** method takes a sign as an input. To prepare the VRML scene for solid viewing by adding normal vectors, you must first call this method, as shown in Listing 7.4.

Listing 7.4 Creating normals.

```
void createNormals(int sign)
{
     normals=0; maxNormals=500;
     normal=new double[maxNormals*6];
     transNormal=new double[maxNormals*6];

    int[] p=new int[500]; //Max vertices in an object
    int nVertices=0;

    double midX=0,midY=0,midZ=0;
     for (int n=0;n<polyVertices;n++)
     {
         int v=poly[n];
          if (v<0)
          {
            //Find the midpoint of the polygon
            midX/=nVertices;
            midY/=nVertices;
            midZ/=nVertices;

            //Find normal
            double vectorX1=model[p[0]*3];
            double vectorY1=model[p[0]*3+1];
            double vectorZ1=model[p[0]*3+2];

            double vectorX2=model[p[1]*3];
            double vectorY2=model[p[1]*3+1];
            double vectorZ2=model[p[1]*3+2];

            double vectorX3=model[p[2]*3];
            double vectorY3=model[p[2]*3+1];
            double vectorZ3=model[p[2]*3+2];

            double px=vectorX1-vectorX2;
            double py=vectorY1-vectorY2;
            double pz=vectorZ1-vectorZ2;

            double qx=vectorX1-vectorX3;
            double qy=vectorY1-vectorY3;
            double qz=vectorZ1-vectorZ3;

            double nx=py*qz-pz*qy;
            double ny=pz*qx-px*qz;
            double nz=px*qy-py*qx;
```

```
            double length=Math.sqrt(nx*nx+ny*ny+nz*nz);
            double scale=1.0/length;

            nx*=sign*scale;
            ny*=sign*scale;
            nz*=sign*scale;

            addNormal(midX,midY,midZ,midX+nx,midY+ny,midZ+nz);

               nVertices=0; midX=0; midY=0; midZ=0;
          }
          else
          {
               p[nVertices++]=v;
               midX+=model[v*3];
               midY+=model[v*3+1];
               midZ+=model[v*3+2];
          }
     }
}
```

Make sure the normal vectors in the scene point the right way, or you might end up with objects turned inside out like the cube in Figure 7.4.

Hidden Surfaces

You now know how to calculate the brightness of each surface in a scene, but before we can create the renderer, we first have to solve another common problem when it comes to computer graphics: the dreaded hidden surface elimination.

There are several ways to make sure surfaces blocked by other surfaces are hidden from the viewer. The most accurate method is the z-buffering technique described earlier. But when you're dealing with large surfaces, z-buffering is far too slow for Java. So let's throw that technique overboard, even though this means we won't be able to correctly display surfaces penetrated by other surfaces. This is not a big problem, though, since we can create all scenes without depending on this feature anyway.

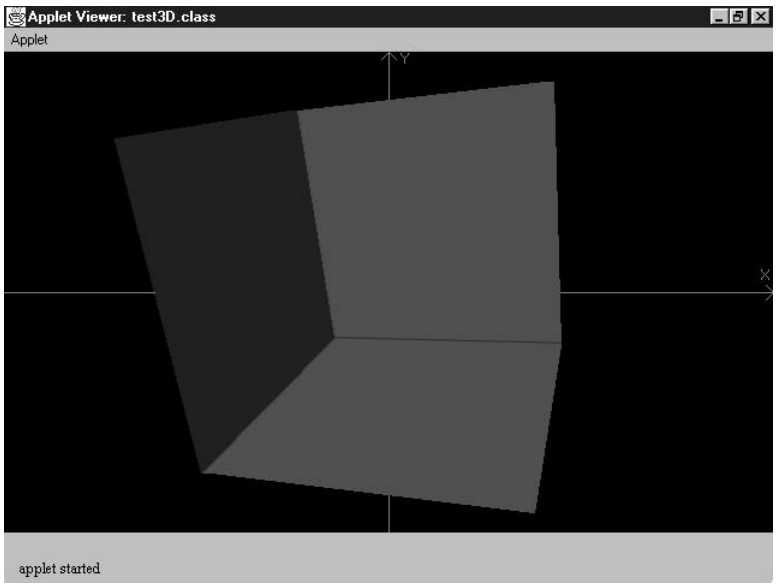

FIGURE 7.4

An inverted cube.

The next technique depends on surface normal vectors being available. Fortunately, in our case, they are. That gives us a very fast hidden surface elimination technique almost for free. The technique asks one simple question before rendering each surface: Is the normal vector of the surface pointing toward or away from the viewer? If the normal vector is pointing away from the viewer, the surface is facing away and isn't visible. This implies that only one side of the surface is visible. In all convex objects, this hidden surface elimination method is all you'll ever need. In the cube example, all invisible faces will be eliminated correctly this way. The angle between the surface's midpoint and its normal vector determines whether the surface is going to be visible. Again, we'll use the dot product of two vectors to determine the angle between them. If **offset** points to the surface's normal vector, the optimized hidden surface algorithm will look like this:

```
double dx=transNormal[offset+3]-transNormal[offset];
double dy=transNormal[offset+3+1]-transNormal[offset+1];
double dz=transNormal[offset+3+2]-transNormal[offset+2];
if ((transNormal[offset]*dx+transNormal[offset+1]*dy
     +transNormal[offset+2]*dz)<=0)
{
     //The surface is visible. Draw it.
}
```

As I already mentioned, this technique is fine when dealing with a scene containing a single convex object. But normally, you want to place more than one object in the scene. In this case, the *painters algorithm* can be used. The painters algorithm simply paints all surfaces in the correct order, from the ones farthest away from the viewer to the ones closest. The theory is simple, but to make it work you need some heavy optimization. First of all, you can remove all self hidden surfaces by using the normal vector technique, which will help a lot to boost the performance. The remaining surfaces must then be added to a list. The list should contain no more than one long value for each surface, to keep the amount of data that needs to be handled to a minimum. In this long value, you can store the distance to the midpoint of the surface, its illumination factor, and its offset using bit shifting, which we discussed in Chapter5.

When the list is finished, you can sort it using QuickSort, a very fast sorting algorithm that was also introduced earlier. Just make sure the high word of each long value is used for storing the distance. This will ensure that the polygons are sorted correctly, depending on their distance from the viewer. When the list is sorted, you can simply traverse it and draw all polygons each element points to in the correct color.

Implementing The Renderer

Now that you have all the basics needed for implementing the solid renderer, the only thing left to mention is that you should add depth shading for increased realism. The distance to each surface from the viewer is always calculated with the painters algorithm anyway, so you can use this value instead of the surface's z coordinate, which would normally be faster. The farther away the surface is, the more faded into the background color it will be.

It's also worth mentioning that the light source can be included as part of the scene. In that case, it will be rotated along with it, or it could be placed relative to the camera. The latter position is the default and is more realistic, since the light reflected off the surfaces will change according to how much the object is rotated. By rotating the light around the scene along with the objects itself, the colors of the surfaces won't change, and in an animation the result would be less interesting. When **cameraLightOn** is true, the light source will be placed relative to the camera. In the other case, it will be placed relative to the scene. The complete solid rendering source is given in Listing 7.5. It's called by passing the render code 3 to the **.render** method.

Listing 7.5 Rendering a solid scene.

```
int polyIndex=0;
//Project vertices
projX=new int[vertices];
projY=new int[vertices];
i=0;

for (int n=0;n<vertices*3;n+=3)
{
     int x=(int)(transModel[n]);
     int y=(int)(transModel[n+1]);
     int z=(int)(transModel[n+2]);
     projX[i]=x*frontPlane/z+centerX;
     projY[i]=-y*frontPlane/z+centerY;
     i++;
}

double lightX;
double lightY;
double lightZ;
if (!cameraLightOn)
{
    lightX=transLight[0];
    lightY=transLight[1];
    lightZ=transLight[2];
}
else
{
    lightX=light[0];
    lightY=light[1];
```

```
    lightZ=light[2]-frontPlane;
}

long[] sortArray=new long[polygons];
int visiblePolygons=0;
int polygonOffset=0;

colorStep=256.0/(backPlane-frontPlane);
p=new Polygon();
v0=poly[0];

for (int n=0;n<polyVertices;n++)
{
     int v=poly[n];
     if (v<0)
     {
         //Polygon closed
          p.addPoint(projX[v0],projY[v0]);
          v0=poly[n+1];
          int offset=polyIndex*6;
        double dx=transNormal[offset+3]-transNormal[offset];
        double dy=transNormal[offset+3+1]-transNormal[offset+1];
        double dz=transNormal[offset+3+2]-transNormal[offset+2];
        if ((transNormal[offset]*dx+transNormal[offset+1]*dy
          +transNormal[offset+2]*dz)<=0)
        {
            //The face of the polygon is headed toward
            //the viewer, so it's visible.

            //Calculate the amount of light the polygon
            //receives from the light source
            double normalx=transNormal[offset+3]-
               transNormal[offset];
            double normaly=transNormal[offset+3+1]-
               transNormal[offset+1];
            double normalz=transNormal[offset+3+2]-
                transNormal[offset+2];

             dx=lightX-transNormal[offset];
             dy=lightY-transNormal[offset+1];
             dz=lightZ-transNormal[offset+2];
             double illumination=(dx*normalx+dy*normaly+
                dz*normalz)/Math.sqrt(dx*dx+dy*dy+dz*dz);

            double xMid=transNormal[offset];
```

```
            double yMid=transNormal[offset+1];
            double zMid=transNormal[offset+2];
            int c=(int)(128*illumination)+128;

            //Store the index and the color of this polygon in
            //the distance-from-camera array to be sorted
            //afterward
            //This value could be replaced with zMid for
            //slightly faster computing, but it's a tad more
            //inaccurate
            long d=(long)(Math.sqrt(xMid*xMid+
               yMid*yMid+zMid*zMid));

            //Depth shading; Simply remove if unwanted
            c-=(int)((d-frontPlane)*colorStep); if (c<0) c=0;

            sortArray[visiblePolygons++]=(d<<32)+
               (c<<16)+polygonOffset;
          }
          polyIndex++;
          p=new Polygon();
          polygonOffset=n+1;
     }
     else
          p.addPoint(projX[v],projY[v]);
}

//Do the sorting
//If you have only one convex object
//You can drop this; the hidden surface
//algorithm using normal vectors will do the job for you
sort(sortArray, 0, visiblePolygons-1);

//Draw polygons in sorted order
for (int n=visiblePolygons-1;n>=0;n--)
{
    long l=sortArray[n];
    //Extract light intensity (0-255)
     int c=(int)((l>>16)&0xff);
     //Extract polygon offset
     i=(int)(l&0xffff);
     p=new Polygon();
     int v=poly[i];
    do
    {
```

```
            p.addPoint(projX[v],projY[v]);
          v=poly[++i];
      } while (v>=0);

          GC.setColor(colorGrades[c]);
       GC.fillPolygon(p);
}
```

Even relatively simple objects like the Imperial T.I.E. fighter (from the movie "Star Wars") in Figure 7.5 will look very realistic when rendered using depth shading and a light source. Drawing an image in the background will also help.

Adding Shadows

When it comes to realism, the light source renderer really kicks ass. But why stop there? For the ultimate in realism, let's add some shadows as well.

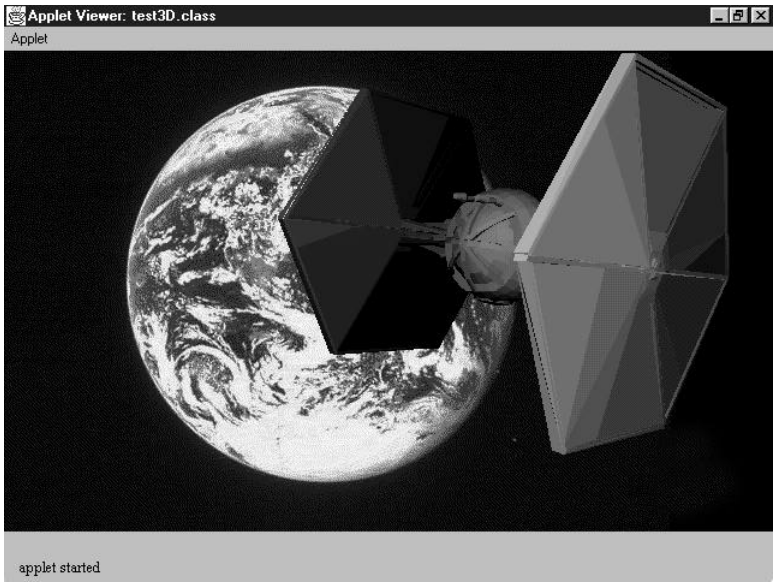

FIGURE 7.5 An imperial T.I.E. fighter.

First we have to figure out what kind of shadows will permit real-time performance. Objects casting shadows on other objects would give the most realistic results, but they would seriously drop the frame-rate. And when you've created a great rotating logo or object, you're probably not interested in cluttering it with dark shadows. But what almost always looks nice is when your objects cast shadows on the background. Imagine your logo rotating in front of a textured background—say an image of a stone wall—while actually casting a shadow on it depending on the angle of the incoming light. This would really create a feeling of depth, and by moving the light around as well, your results will be very impressive. And the good news is that this way of creating shadows is very fast when implemented the way I'm going to show you. If fact, real-time performance with simple objects is no problem at all, even without any JIT.

So how do you calculate the shadow of an object? In theory it's very simple, especially when the shadow is only being cast on a flat surface. Just imagine a line passing through each vertex in the object and the light source. Where this line hits the plane of the background, a shadow vertex is formed. You know all the coordinates of the object's vertices, along with the coordinates of the light. In addition, you know the z coordinate of the background plane. Now all you need to do is find the line's equation for each vertex and solve it for the background's z coordinate. You can then draw the projected shadow polygons using a color darker than the background. Just keep in mind that you need to consider all surfaces when calculating the shadow, since even hidden surfaces cast shadows.

As you can see in Figure 7.6, the task of projecting the shadow is very simple. Just find the equation of the line passing through a vertex and the light source. The equation is then solved for the background's z coordinate.

So why not add a method to the **Model3D** class that draws the shadow for you, given the distance to the background image and the color of the shadow, as shown in Listing 7.6?

Listing 7.6 Rendering a shadow.

```
void renderShadow(double distance, Color c)
{
    //Translate z coordinate of background
```

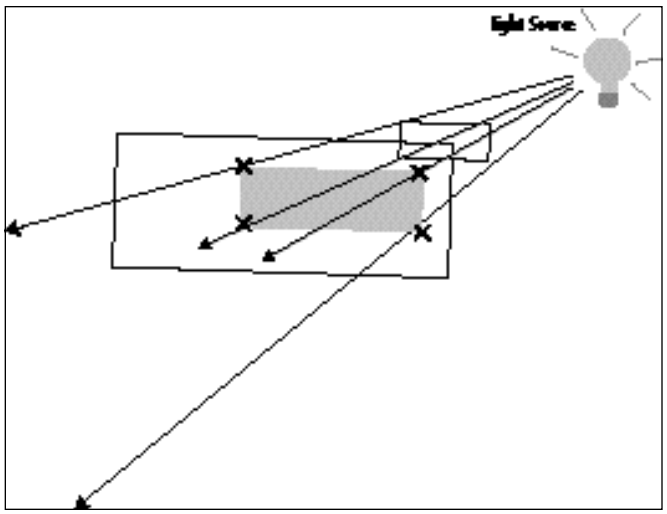

FIGURE 7.6 Projecting the shadow's vertices.

```
distance+=-cameraZ+frontPlane;
 GC.setColor(c);

double lightX;
double lightY;
double lightZ;
if (!cameraLightOn)
{
    lightX=transLight[0];
    lightY=transLight[1];
    lightZ=transLight[2];
}
else
{
    lightX=light[0];
    lightY=light[1];
    lightZ=light[2]-frontPlane;
}


 //Project vertices
 int[] projX=new int[vertices];
 int[] projY=new int[vertices];
 int i=0;
double s=frontPlane/distance;
 for (int n=0;n<vertices*3;n+=3)
 {
```

```
        //Find the intersection point between the background
        //plane and the line defined by the vertex and the light
        //coordinates
        double shadowX=((transModel[n]-lightX)*(distance-
          lightZ))/(transModel[n+2]-lightZ)+lightX;
        //Store the projected shadow vertices
        double shadowY=((transModel[n+1]-lightY)*(distance-
           lightZ))/(transModel[n+2]-lightZ)+lightY;
        projX[i]=(int)(shadowX*s)+centerX;
        projY[i]=(int)(-shadowY*s)+centerY;

          i++;
     }

    //Draw shadow
     Polygon p=new Polygon();
     int v0=poly[0];
     for (int n=0;n<polyVertices;n++)
     {
          int v=poly[n];
          if (v<0)
          {
               p.addPoint(projX[v0],projY[v0]);
               v0=poly[n+1];
               GC.fillPolygon(p);
               p=new Polygon();
          }
          else
               p.addPoint(projX[v],projY[v]);
     }
}
```

Figure 7.7 shows a teapot rendered with shadows, in front of a background image. As you can see, the shadows bring some real depth and realism to the scene. And the best thing is that it's fast!

You must call the **.renderShadow** method prior to the **.render** method, of course, to keep the shadow in the background. The **distance** parameter must be carefully chosen to make sure the plane of shadow projection always is located behind the vertex farthest away. If the object is located even partly behind this plane, the shadow won't be rendered correctly since no time is wasted on clipping the object against this plane. The higher the

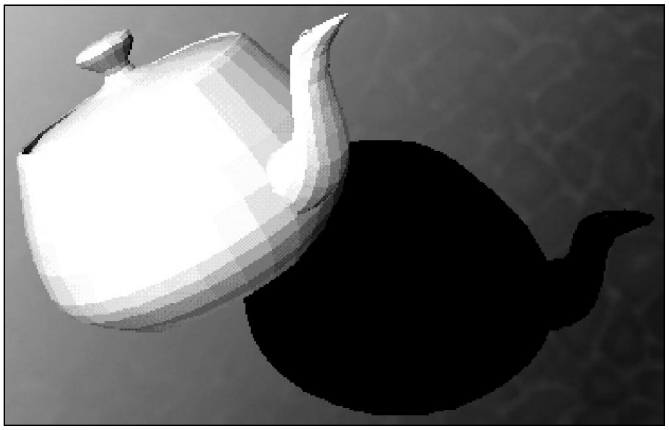

FIGURE 7.7

A shaded teapot.

distance value, the longer the distance will seem to the background plane, and the bigger the background image will appear. The best thing is to keep the distance as short as possible without the background image cutting through the model.

3D Models On The Fly

Although using VRML models is best if you're looking for the ultimate in realism, there are times when you can't bother setting up the scene this way. If you simply want a rotating logo, why should you need to do more than load your logo using **.getImage** and then translate it into a 3D object automatically? Why indeed....

The task of converting an image to a 3D model is very complicated. In fact, a whole book could be written about this subject alone, and it's way beyond the scope of this one. But I feel there are a lot of non-VRML folks who really need another way of creating 3D models, so why not give it a shot?

The most straightforward method of creating a 3D model from an image is to make each pixel in the image into its own 3D cube in the appropriate color. To do this, we'll have to implement a renderer that shows surfaces in all colors. This means we'll have to create a new color object for each surface to be drawn, which is something Java is not particular fast at. This rendering method should, therefore, only be used when absolutely necessary.

If you've read the chapters about image processing, the task of converting the image to 3D cubes should be easy for you. First, you have to make sure the image is finished loading, then you can grab its pixels by using the **PixelGrabber** class. Then you simply traverse the pixels one by one and add a cube whenever the color of the pixel differs from the background color. By default, the top-left color in the image is considered to be the background color. Listing 7.7 is all you need to convert an image to a 3D model using cubes with different colors.

Listing 7.7 Creating a 3D object from an image.

```
void imageTo3D(Image picture)
{
     //First make sure the image is fully loaded
     waitForImage(picture);

     //Get the width and height of the image
     int w=picture.getWidth(this);
     int h=picture.getHeight(this);

    //Allocate buffer to hold the image's pixels
     int[] pixels = new int[w * h];

     //Grab pixels
     PixelGrabber pg = new PixelGrabber(picture,
          0, 0, w, h, pixels, 0, w);
     try {
          pg.grabPixels();
     } catch (InterruptedException e);


    //Color to be considered the background color is
    //located in the top-left corner.
    int bg=pixels[0];
    int cubeSize=200;
    for (int y=0;y<h;y++)
        for (int x=0;x<w;x++)
        {

            int col=pixels[y*w+x];
            if (bg!=col)
            {
                //Add cube to scene if the pixel is not
                //transparent
```

```
                int r=(col&0xff0000)>>16;
                int g=(col&0xff00)>>8;
                int b=col&0xff;
                Color c=new Color(r,g,b);

                 addCube((x-w/2)*cubeSize,(h/2-y)*cubeSize,0,
                   cubeSize/2.2,c);
            }
        }
}
```

Creating 3D logos is very simple using the **.imageTo3D** method. If you choose your image wisely, you'll get good results, or you could just write some text in an image, as in Figure 7.8, if you simply want to produce some 3D very fast.

If you really don't want to mess around with anything but a simple image to create some 3D action on your page, a pretty general program that accepts an image name as an HTML parameter and then converts it to a 3D object before starting to rotate it is shown in Listing 7.8.

FIGURE 7.8

A 3D object converted from a flat image.

Listing 7.8 A pretty general 3D image viewer.

```
import java.io.InputStream;
import java.net.URL;
import java.awt.*;
import java.awt.image.*;

public class image3D extends java.applet.Applet implements Runnable {

    Thread animationThread = null;
    int delay=100,xsize,ysize,ticker;
    Image backBuffer;
    Graphics backGC;
    Model3D model;
    Image background;

    public void init()
    {
        xsize=size().width;
        ysize=size().height;

        //Create virtual screen for double buffering
        backBuffer = createImage(xsize, ysize);
        backGC = backBuffer.getGraphics();

        model=new Model3D(backGC,xsize,ysize);

        String imageName=getParameter("image");
        if (imageName!=null)
        {
              Image picture=getImage(getCodeBase(),imageName);
              //Make 3D cubes out of image
              model.imageTo3D(picture);
              //Scale image to fit into the applet window
             model.autoScale();
             //Always remember to recalculate all normals
             //after an autoScale
             model.createNormals(1);
            model.setLight(-400,200,-100);
          }
    }

    //This method is called when the applet becomes visible
    public void start()
    {
        //Create a thread and start it
```

```
    if(animationThread == null)
    {
        animationThread = new Thread(this);
        animationThread.start();
    }
}

//This method is called when the applet becomes invisible
public void stop()
{
    //Stop animation thread
    animationThread = null;
}

public void run()
{
    long time = System.currentTimeMillis();
    while (animationThread != null)
    {
        //Make sure the frame rate is constant by using the
         //system clock
        try
        {
            time += delay;
            Thread.sleep(Math.max(0,time -
               System.currentTimeMillis()));
        }
        catch (InterruptedException e);
        //Initiate a repaint of the screen
        repaint();
    }
}

public void paint(Graphics g)
{
        //Clear screen
        //backGC.drawImage(background, 0, 0, this);
        backGC.setColor(Color.lightGray);
        backGC.fillRect(0,0,xsize,ysize);

        ticker++;
        model.rotateXYZ(ticker*3,ticker*2,ticker);
        model.renderShadow(300,Color.gray);
        model.render(4);

        g.drawImage(backBuffer, 0, 0, this);
```

```
    }

    public void update(Graphics g)
    {
        paint(g);
    }
}
```

The HTML file for the program in Listing 7.8 looks like this:

```
<HTML>
<APPLET CODE="image3D.class" WIDTH=400 HEIGHT=300>
<param name="image" value="logo.gif">
</APPLET>
</HTML>
```

A General 3D Viewer

Why program anything at all when you want to show off some 3D? You can create a fairly general 3D viewer using the **Model3D** class. All you need to do is make it accept the proper HTML parameters. This viewer will be much like the one presented in the previous chapter, but now there is support for shadows, a background image, and light-sourced 3D. Because most interesting objects are too complicated for real-time viewing, all the frames in the animation are pregenerated and then displayed in a loop.

Listing 7.9 shows the HTML file for the VRML viewer.

Listing 7.9 HTML file for the general 3D viewer.

```
<HTML>
<APPLET CODE="animate3D.class" WIDTH=200 HEIGHT=200>
<param name="model" value="hammer.wrl">

<param name="rotate dx" value="0">
<param name="rotate dy" value="360">
<param name="rotate dz" value="0">

<param name="frames" value="30">

<param name="rotate x" value="45">
<param name="rotate y" value="15">
<param name="rotate z" value="0">
```

```
<param name="camera x" value="0">
<param name="camera y" value="0">
<param name="camera z" value="-250">
<param name="mode" value="3">

<param name="bgRed" value="0">
<param name="bgGreen" value="0">
<param name="bgBlue" value="0">

<param name="Red" value="255">
<param name="Green" value="255">
<param name="Blue" value="255">

<param name="light x" value="-400">
<param name="light y" value="800">
<param name="light z" value="0">

<param name="background" value="bakgrunn.jpg">
<param name="normalSign" value="1">

<param name="shadowPlane" value="300">
</APPLET>
</HTML>
```

Let's review the different parameters in the HTML file in Listing 7.9:

- **model**—The name of the VRML file to be displayed.
- **rotate dx**, **rotate dy**, **rotate dz**—The number of degrees the object is to spin about the respective axes during the animation. This number should be divisible by 360 to allow for looped animation.
- **Frames**—The number of frames to be pregenerated. The more frames rendered, the smoother the animation will be.
- **rotate x**, **rotate y**, **rotate z**—The number of degrees the object is to be rotated about the respective axes during initialization.
- (**camera x**, **camera y**, **camera z**)—The location of the camera.
- **mode**—The rendering mode:

  0 = Draw depth shaded vertices.

  1 = Draw polygons using single colored lines.

2 = Draw polygons using depth shaded lines.

3 = Draw filled polygons with light sources and depth shading.

4 = Same as 3, but now each polygon may be rendered in its own color.

- **bgRed, bgGreen**, **bgBlue**—The background's color components.
- **Red**, **Green**, **Blue**—The color of the model.
- (**light x**, **light y**, **light z**)—The coordinate of the light source.
- **background**—The name of the background image.
- **normalSign**—The sign of all calculated normal vectors. If the object seems inside out, you can change this sign to fix it. It can be -1 or 1.
- **shadowPlane**—The z coordinate of the background image. This is used if you want the object to cast shadows. Make sure this value is large enough so the background plane does not cut through the 3D model.

If you leave out some of these parameters, the default values will be used instead. For example, you don't *have* to use a background image or shadows. So let's take a closer look at the HTML interpreter: the Java program itself. It's very simple, and in fact most of the work is done by the **Model3D** class. Listing 7.10 shows the program for general animated viewing of 3D objects.

Listing 7.10 A general VRML viewer.

```
import java.io.InputStream;
import java.net.URL;
import java.awt.*;
import java.awt.image.*;

public class animate3D extends java.applet.Applet implements Runnable {

     Thread animationThread = null;
     int delay=120,xsize,ysize;
     int xRot,yRot,zRot;
     int rotDx,rotDy,rotDz;
```

```
int mode,frames=1,frameCounter;
Image backBuffer,background;
Graphics backGC;
Model3D model;
Image[] frame;
double shadowPlane=-1;
Color bgColor,color;

public void init()
{
     xsize=size().width;
     ysize=size().height;

     //Create virtual screen for double buffering
     backBuffer = createImage(xsize, ysize);
     backGC = backBuffer.getGraphics();

     //Read HTML parameters
     String modelName=getParameter("model");

     //Read sign of normal vector
     int sign=1;
     String s=getParameter("normalSign");
     if (s!=null) sign = Integer.parseInt(s);

     //Initial tilt
     String xt=getParameter("rotate x");
     String yt=getParameter("rotate y");
     String zt=getParameter("rotate z");
     if (xt!=null) xRot = Integer.parseInt(xt);
     if (yt!=null) yRot = Integer.parseInt(yt);
     if (zt!=null) zRot = Integer.parseInt(zt);

     //Rotation movement
     String xr=getParameter("rotate dx");
     String yr=getParameter("rotate dy");
     String zr=getParameter("rotate dz");
     if (xr!=null) rotDx = Integer.parseInt(xr);
     if (yr!=null) rotDy = Integer.parseInt(yr);
     if (zr!=null) rotDz = Integer.parseInt(zr);

     String f=getParameter("frames");
     if (f!=null) frames = Integer.parseInt(f);

     //Camera
     String cx=getParameter("camera x");
```

```
String cy=getParameter("camera y");
String cz=getParameter("camera z");
int cameraX=0,cameraY=0,cameraZ=0;
if (cx!=null) cameraX = Integer.parseInt(cx);
if (cy!=null) cameraY = Integer.parseInt(cy);
if (cz!=null) cameraZ = Integer.parseInt(cz);

//Light source
String lx=getParameter("light x");
String ly=getParameter("light y");
String lz=getParameter("light z");
int lightX=0,lightY=0,lightZ=0;
if (lx!=null) lightX = Integer.parseInt(lx);
if (ly!=null) lightY = Integer.parseInt(ly);
if (lz!=null) lightZ = Integer.parseInt(lz);

//Background color
String bgR=getParameter("bgRed");
String bgG=getParameter("bgGreen");
String bgB=getParameter("bgBlue");
int bgRed=0,bgGreen=0,bgBlue=0;
if (bgR!=null) bgRed = Integer.parseInt(bgR);
if (bgG!=null) bgGreen = Integer.parseInt(bgG);
if (bgB!=null) bgBlue = Integer.parseInt(bgB);
bgColor=new Color(bgRed,bgGreen,bgBlue);

//Foreground color
String r=getParameter("red");
String g=getParameter("green");
String b=getParameter("blue");
int red=255,green=255,blue=255;
if (r!=null) red = Integer.parseInt(r);
if (g!=null) green = Integer.parseInt(g);
if (b!=null) blue = Integer.parseInt(b);
color=new Color(red,green,blue);

String m=getParameter("mode");
if (m!=null) mode = Integer.parseInt(m);

String shadow=getParameter("shadowPlane");
if (shadow!=null) shadowPlane = Integer.parseInt(shadow);

showStatus("Loading Model");
//Load model
try
{
```

```
            InputStream is=new URL(getCodeBase(),
                modelName).openStream();
            model=new Model3D(is,backGC,xsize,ysize);
        }
        catch(Exception e){
            String a=e.toString();
            System.out.println(a);
        };

        String bg=getParameter("background");
        if (bg!=null)
        {
            background=getImage(getCodeBase(),bg);
            model.waitForImage(background);
       }

        //Optional, might slow things down depending
        //on the model and the number of frames
        /*if (mode==2)
        {
            showStatus("Optimizing Model");
            model.optimizeModel();
        }*/

        //Move camera
        model.moveCamera(cameraX,cameraY,cameraZ);

        //Set light
        model.setLight(lightX,lightY,lightZ);

      //Create normal vectors
        model.createNormals(sign);

        //Set color
        model.setBgColor(bgColor);
        model.setColor(color);

        frame=new Image[frames];
    }

    //This method is called when the applet becomes visible
    public void start()
    {
        //Create a thread and start it
        if(animationThread == null)
        {
            animationThread = new Thread(this);
```

```
            animationThread.start();
        }
    }

    //This method is called when the applet becomes invisible
    public void stop()
    {
        //Stop animation thread
        animationThread = null;
    }

    public void run()
    {
        long time = System.currentTimeMillis();
        while (animationThread != null)
        {
            //Make sure the frame rate is constant by using
            //the system clock
            try
            {
                time += delay;
                Thread.sleep(Math.max(0,time -
                    System.currentTimeMillis()));
            }
            catch (InterruptedException e);
            //Initiate a repaint of the screen
            repaint();
        }
    }

    public void paint(Graphics g)
    {
        int n=frameCounter%frames;

        if (frame[n]==null)
          {
            if (n<frames-1)
                showStatus("Rendering frame "+n);
            else
                showStatus("Finished rendering");
           if (background!=null)
               backGC.drawImage(background,0,0,this);
           else
           {
```

```
                backGC.setColor(bgColor);
                backGC.fillRect(0,0,xsize,ysize);
            }

            model.rotateXYZ((rotDx*n)/frames+xRot,(rotDy*n)/
               frames+yRot,(rotDz*n)/frames+zRot);
          if (shadowPlane>0)
            model.renderShadow(shadowPlane,Color.black);
            model.render(mode);
            frame[n]=createImage(xsize,ysize);
            Graphics GC = frame[n].getGraphics();
            GC.drawImage(backBuffer, 0, 0, this);
        }
        else
            backGC.drawImage(frame[n],0,0,this);

        g.drawImage(backBuffer, 0, 0, this);
        frameCounter++;
    }

    public void update(Graphics g)
    {
        paint(g);
    }

}
```

Using the general 3D viewer in Listing 7.10, you can quickly make your VRML objects rotate and cast shadows in a light sourced environment, like in Figure 7.9.

Texture Mapping

When some serious realism is needed, the number-one 3D technique is texture mapping. This method lets you map any image onto a 3D surface. Sadly, Java without any JIT is not really capable of handling such advanced 3D in real-time, but let's not let facts get in the way! By prerendering texture-mapped models, you can play them back fast later. Imagine, for example, a precalculated rotating cube. You could rotate it around the y axis using, say, 45 frames. The user could then interactively rotate the cube in the direction and at the speed he chooses. This cube could be a 3D menu for example, with a different option on each side, and demonstrates one of the many useful things texture mapping can be used for.

FIGURE 7.9

A rotating 3D hammer.

This book focuses more on how to generate flying logos using texture mapping techniques. As you learned earlier, you can easily convert an image to 3D by making a cube out of each pixel it contains. Although this is a handy technique, it also means that each pixel is actually converted into an object with six different surfaces when only one is needed, and a lot of space and time are wasted. A much better solution is to create a grid of polygons, all attached together and formed by four vertices. This really starts to be interesting if you apply a different color to each of the surfaces in this grid. For example, you can easily apply the colors from the pixels forming an image.

After converting the image to 3D like this, you can do all kinds of transformations to it. It can be bathed in light, moved, scaled, and rotated. But before you start to render the image object, you can do even more interesting things. How about giving each pixel a depth according to how dark or light it is? This would generate an object with curves, just like a landscape. This is a perfect way of creating a 3D model out of a logo, and if you use the right colors it will probably be more astonishing than any VRML object.

Even though this technique provides visually stunning results, it's fairly easy to implement. All you need to do is first build a grid of vertices to hold all the pixels in the image. You need as many vertices as there are pixels. When this is done, you simply traverse the image and create a polygon for each of the pixels encountered. Of course, the color of each polygon is set to that of the pixel it's supposed to represent. Listing 7.11 gives the details of how to do this.

Listing 7.11 Texture mapping.

```
void imageTo3DMap(Image picture,double height)
{
     //First make sure the image is fully loaded
     waitForImage(picture);

     //Get the width and height of the image
     int w=picture.getWidth(this);
     int h=picture.getHeight(this);

    //Allocate buffer to hold the image's pixels
     int[] pixels = new int[w * h];

     //Grab pixels
     PixelGrabber pg = new PixelGrabber(picture,
        0, 0, w, h, pixels, 0, w);
     try {
          pg.grabPixels();
     } catch (InterruptedException e);


    //Set up all vertices
    double scale=(w+h)/2;
    int offset=vertices;
    for (int y=0;y<h;y++)
        for (int x=0;x<w;x++)
        {
            int col=pixels[y*w+x];
            int r=(col&0xff0000)>>16;
            int g=(col&0xff00)>>8;
            int b=col&0xff;
            double gray=(r+g+b)/3.0;
            addVertex ((x-w/2),(h/2-y),-gray*height*scale/1000);
        }
```

```
    //Set up polygons
    int[] p=new int[4];
    for (int y=0;y<h-1;y++)
        for (int x=0;x<w-1;x++)
        {
            p[0]=offset+x+y*w;
            p[1]=offset+(x+1)+y*w;
            p[2]=offset+(x+1)+(y+1)*w;
            p[3]=offset+x+(y+1)*w;
            int col=pixels[y*w+x];
            addPolygon (p,new Color(col));
        }
}
```

By mapping an image to a grid of surfaces and adding depth information from each pixel's intensity value, the result is a landscape-like image, as shown in Figure 7.10.

The method in Listing 7.11 accepts two parameters. One is the picture to be converted, and the other is the depth scalar. The higher the depth scalar is, the bigger curves the image will get. If the depth scalar is 0, the

FIGURE 7.10

An image texture mapped to a 3D object.

image will simply appear flat. Rotating flat images can also come in handy, so keep this trick in mind.

You can even implement a general viewer for this kind of object. It can be built on the previous viewer, and will accept one more parameter, namely **mapDepth**. It's simply the depth scalar mentioned earlier. Some parameters are also let out, like the ones for setting the foreground color, since they would no longer make any sense. Listing 7.12 shows the general texture mapped 3D object viewer.

Listing 7.12 A general 3D image viewer using texture mapping.

```
import java.awt.*;
import java.awt.image.*;

public class imageMap3D extends java.applet.Applet implements Runnable {

     Thread animationThread = null;
     int delay=120,xsize,ysize;
     int xRot,yRot,zRot;
     int rotDx,rotDy,rotDz;
     int mode,frames=1,frameCounter;
     Image backBuffer,background;
     Graphics backGC;
     Model3D model;
     Image[] frame;
     double shadowPlane=-1,mapDepth;
     Color bgColor;

     public void init()
     {
          xsize=size().width;
          ysize=size().height;

          //Create virtual screen for double buffering
          backBuffer = createImage(xsize, ysize);
          backGC = backBuffer.getGraphics();

          //Read HTML parameters
          String modelName=getParameter("model");

          //Initial tilt
          String xt=getParameter("rotate x");
          String yt=getParameter("rotate y");
          String zt=getParameter("rotate z");
```

```
if (xt!=null) xRot = Integer.parseInt(xt);
if (yt!=null) yRot = Integer.parseInt(yt);
if (zt!=null) zRot = Integer.parseInt(zt);

//Rotation movement
String xr=getParameter("rotate dx");
String yr=getParameter("rotate dy");
String zr=getParameter("rotate dz");
if (xr!=null) rotDx = Integer.parseInt(xr);
if (yr!=null) rotDy = Integer.parseInt(yr);
if (zr!=null) rotDz = Integer.parseInt(zr);

String f=getParameter("frames");
if (f!=null) frames = Integer.parseInt(f);

String h=getParameter("mapDepth");
if (h!=null) mapDepth = Integer.parseInt(h);

//Camera
String cx=getParameter("camera x");
String cy=getParameter("camera y");
String cz=getParameter("camera z");
int cameraX=0,cameraY=0,cameraZ=0;
if (cx!=null) cameraX = Integer.parseInt(cx);
if (cy!=null) cameraY = Integer.parseInt(cy);
if (cz!=null) cameraZ = Integer.parseInt(cz);

//Light source
String lx=getParameter("light x");
String ly=getParameter("light y");
String lz=getParameter("light z");
int lightX=0,lightY=0,lightZ=0;
if (lx!=null) lightX = Integer.parseInt(lx);
if (ly!=null) lightY = Integer.parseInt(ly);
if (lz!=null) lightZ = Integer.parseInt(lz);

//Background color
String bgR=getParameter("bgRed");
String bgG=getParameter("bgGreen");
String bgB=getParameter("bgBlue");
int bgRed=0,bgGreen=0,bgBlue=0;
if (bgR!=null) bgRed = Integer.parseInt(bgR);
if (bgG!=null) bgGreen = Integer.parseInt(bgG);
if (bgB!=null) bgBlue = Integer.parseInt(bgB);
bgColor=new Color(bgRed,bgGreen,bgBlue);
```

```
     String shadow=getParameter("shadowPlane");
     if (shadow!=null) shadowPlane = Integer.parseInt(shadow);

     showStatus("Loading Model");
     //Load model
        Image picture=getImage(getCodeBase(),modelName);
     model=new Model3D(backGC,xsize,ysize);
     model.imageTo3DMap(picture,mapDepth);
     model.autoScale();

     String bg=getParameter("background");
     if (bg!=null)
     {
         background=getImage(getCodeBase(),bg);
         model.waitForImage(background);
    }

     //Move camera
     model.moveCamera(cameraX,cameraY,cameraZ);

     //Set light
     model.setLight(lightX,lightY,lightZ);

   //Re-create normal vectors after autoScale
     model.createNormals(1);

     //Set color
     model.setBgColor(bgColor);

     frame=new Image[frames];
}

//This method is called when the applet becomes visible
public void start()
{
     //Create a thread and start it
     if(animationThread == null)
     {
          animationThread = new Thread(this);
          animationThread.start();
     }
}

//This method is called when the applet becomes invisible
public void stop()
{
     //Stop animation thread
```

```
        animationThread = null;
    }

    public void run()
    {
        long time = System.currentTimeMillis();
        while (animationThread != null)
        {
            //Make sure the frame rate is constant by using
            //the system clock
            try
            {
                time += delay;
                Thread.sleep(Math.max(0,time -
                    System.currentTimeMillis()));
            }
            catch (InterruptedException e);
            //Initiate a repaint of the screen
            repaint();
        }
    }

    public void paint(Graphics g)
    {
        int n=frameCounter%frames;

        if (frame[n]==null)
          {
            if (n<frames-1)
                showStatus("Rendering frame "+n);
            else
                showStatus("Finished rendering");
           if (background!=null)
               backGC.drawImage(background,0,0,this);
           else
           {
                backGC.setColor(bgColor);
                backGC.fillRect(0,0,xsize,ysize);
            }

            model.rotateXYZ((rotDx*n)/frames+xRot,(rotDy*n)/
               frames+yRot,(rotDz*n)/frames+zRot);
          if (shadowPlane>0)
              model.renderShadow(shadowPlane,Color.black);
            model.render(5);
```

```
                frame[n]=createImage(xsize,ysize);
                Graphics GC = frame[n].getGraphics();
                GC.drawImage(backBuffer, 0, 0, this);
           }
           else
                backGC.drawImage(frame[n],0,0,this);

           g.drawImage(backBuffer, 0, 0, this);
           frameCounter++;
      }

      public void update(Graphics g)
      {
           paint(g);
      }
}
```

You might have noticed that rendering mode 5 is used in Listing 7.12. This mode makes sure no surfaces are eliminated using the normal vector method. It also ensures that the curved surface has a visible backside. If you try to render this kind of object using mode 4, the surfaces facing away from the viewer will be eliminated, which makes the object full of holes unless it's completely flat.

Try experimenting a bit with various logos to find out which are most effectively translated to 3D. Not all images will look that pretty. It's often a good idea to use images without sharp contrasts. It's also worth mentioning that you can combine normal objects and texture mapped objects. If you load a VRML scene, you can easily add a texture mapped surface to it if you want.

The Final Model3D Class

The Model3D class has been through quite a few changes in this chapter, and for your convenience, Listing 7.13 shows it in all its glory.

Listing 7.13 The final Model3D class.

```
import java.io.*; //StreamTokenizer;
import java.awt.*;
import java.awt.image.*;

/** The Model3D class **/
class Model3D extends java.applet.Applet
{
     int backPlane=1500,frontPlane=500;
```

```
int vertices,maxVertices=500;
int normals,maxNormals=500;
int polyVertices,maxPolyVertices=500,polygons,fastLines;
int width,height,centerX,centerY;
double origoX=0,origoY=0,origoZ=0;
double cameraX=0,cameraY=0,cameraZ=0;
int[] zBuffer,zBufferReset,screenBuffer,screenBufferReset;
double[] model,transModel,normal,transNormal,light,transLight;
int[] poly,fastLine;
Graphics GC;
Matrix3D mat;
Color[] colorGrades;
int[] colorGrades2;
Color color,bgColor;
boolean cameraLightOn=true;

Model3D (Graphics g, int xsize, int ysize)
{
     initialize(g,xsize,ysize);
}

Model3D (InputStream is, Graphics g, int xsize, int ysize)
                  throws IOException
{
     initialize(g,xsize,ysize);

     StreamTokenizer st = new StreamTokenizer(is);
     st.whitespaceChars(32,44); // space until comma
     st.eolIsSignificant(false);
     st.commentChar('#');

     int ttype=0,vertAdded=0,polyOffset=0,nPoly=0;
     do {
          ttype=st.nextToken();

          switch (ttype) {

               case StreamTokenizer.TT_WORD:

                    if ("point".equals(st.sval))
                    {
                        st.nextToken();
                           st.nextToken();
                        do
                         {
```

```
                    double x = 0, y = 0, z = 0;

                     x = st.nval;

                st.nextToken();
                if ((st.ttype ==
                     StreamTokenizer.TT_WORD)&&
                     (st.sval!=null))
                {
                    st.nextToken();
                    x=0;
                }

                     y = st.nval;

                st.nextToken();
                if ((st.ttype ==
                     StreamTokenizer.TT_WORD)&&
                     (st.sval!=null))
                {
                    st.nextToken();
                    y=0;
                }

                          z = st.nval;

                st.nextToken();
                if ((st.ttype ==
                     StreamTokenizer.TT_WORD)&&
                     (st.sval!=null))
                {
                    st.nextToken();
                    z=0;
                }

                      addVertex( x, y, z);
                      vertAdded++;

         } while(st.ttype == StreamTokenizer.TT_NUMBER);
}
else if ("coordIndex".equals(st.sval))
{
     while (st.ttype!=
        StreamTokenizer.TT_NUMBER) st.nextToken();
     while (st.ttype == StreamTokenizer.TT_NUMBER)
     {
```

```
                    int p=(int)(st.nval);
                    if (p>=0)
                    {
                         addPolyVert(p+polyOffset);
                         nPoly++;
                    }
                    else
                    {
                         addPolyVert(p);
                         polygons++;
                    }
                    st.nextToken();
               }
          }
          else if (("TransformSeparator".equals(st.sval))
                    ||("Separator".equals(st.sval)))
          {
               polyOffset=vertAdded;
               st.nextToken();
          }

          default:
               break;
          }
     } while (ttype!=StreamTokenizer.TT_EOF);
     is.close();
     autoScale();

     fastLines=nPoly*2;
     fastLine=new int[fastLines];
     int i=0, n=0, v1, v2, v0=poly[0];
     do
     {
          v1=poly[n];
          v2=poly[n+1];
          if (v2<0)
          {
               v2=v0;
               v0=poly[n+2];
               n++;
          }
          fastLine[i++]=v1;
          fastLine[i++]=v2;
          n++;
     } while (n<polyVertices);
```

```
}

void optimizeModel()
{
     //Eliminate overlapping lines
     int[] fastLine2=new int[fastLines];
     int fastLines2=0;
     for (int n=0;n<fastLines;n+=2)
     {
          int i1=fastLine[n];
          int i2=fastLine[n+1];
          boolean exists=false;
          for (int i=0;i<fastLines2;i+=2)
          {
               if ((fastLine2[i]==i1)&&(fastLine2[i+1]==
                    i2)) exists=true;
               if ((fastLine2[i]==i2)&&(fastLine2[i+1]==
                    i1)) exists=true;
          }
          if (!exists)
          {
               fastLine2[fastLines2]=i1;
               fastLine2[fastLines2+1]=i2;
               fastLines2+=2;
          }
     }
     fastLine=new int[fastLines2];
     System.arraycopy(fastLine2,0,fastLine,0,fastLines2);
     fastLines=fastLines2;
}

void autoScale()
{
     if (vertices>0)
     {
          //First, find vertex farthest away from the
          //center
          double maxDist=0;
          for (int n=3;n<vertices*3;n+=3)
          {
               double x=model[n];
               double y=model[n+1];
               double z=model[n+2];
               double dist=Math.sqrt(x*x+y*y+z*z);
               if (dist>maxDist) maxDist=dist;
          }
```

```
            //Find scale factor
            double scale=Math.min(width,height)/maxDist;

            for (int n=0;n<vertices*3;n+=3)
            {
                 model[n]*=scale;
                 model[n+1]*=scale;
                 model[n+2]*=scale;
            }

            for (int n=0;n<normals*3*2;n+=3)
            {
                 normal[n]*=scale;
                 normal[n+1]*=scale;
                 normal[n+2]*=scale;
            }

            //Make sure the object can be rotated around any
            //axis without passing through the front clipping
            //plane
            setCamera(0,0,-(maxDist*scale));

            System.arraycopy(model,0,transModel,0,vertices*3);
       }
  }

  void setCameraLightOn(boolean state)
  {
      cameraLightOn=state;
  }

  void initialize(Graphics g,int xsize,int ysize)
  {
       GC=g;
       width=xsize;
       height=ysize;
       centerX=Math.round(width/2);
       centerY=Math.round(height/2);
       mat=new Matrix3D();

       //Init z-buffer and screen-buffer
       zBuffer=new int[width*height];
       zBufferReset=new int[width*height];
       screenBuffer=new int[width*height];
```

```
        screenBufferReset=new int[width*height];

        //Init default colors
        setBgColor(Color.black);
        setColor(Color.white);

        for (int n=0;n<width*height;n++)
        {
             zBufferReset[n]=backPlane;
        }
        System.arraycopy(zBufferReset,0,zBuffer,0,width*height);
        System.arraycopy(screenBufferReset,0,
            screenBuffer,0,width*height);

        model=new double[maxVertices*3];
        transModel=new double[maxVertices*3];
        normal=new double[maxNormals*6];
        transNormal=new double[maxNormals*6];

        light=new double[3];
        transLight=new double[3];
        light[0]=-frontPlane/2;
        light[1]=frontPlane;
        light[2]=-frontPlane;

        poly=new int[maxPolyVertices];
    }

    void setBgColor(Color c)
    {
        bgColor=new Color(c.getRed(),c.getGreen(),c.getBlue());
        int ARGB=0xff000000+(c.getRed()<<16)+
                 (c.getGreen()<<8)+(c.getBlue());
        for (int n=0;n<width*height;n++)
        screenBufferReset[n]=ARGB;
    }

    void setColor(Color c)
    {
        int r1=c.getRed();
        int g1=c.getGreen();
        int b1=c.getBlue();

        int r2=bgColor.getRed();
        int g2=bgColor.getGreen();
        int b2=bgColor.getBlue();
```

```
        color=new Color(r1,g1,b1);

        //Create color tones
        colorGrades=new Color[256];
        colorGrades2=new int[256];
        for (int n=0;n<256;n++)
        {
            int rc=(r1*n)/256+(r2*(255-n))/256;
            int gc=(g1*n)/256+(g2*(255-n))/256;
            int bc=(b1*n)/256+(b2*(255-n))/256;
            colorGrades[n]=new Color(rc,gc,bc);
            colorGrades2[255-n]=(0xff000000)+(rc<<16)+(gc<<8)+bc;
        }
    }

    void addVertex(double x, double y, double z)
    {
        if (vertices<maxVertices)
        {
            model[vertices*3]=x;
            model[vertices*3+1]=y;
            model[vertices*3+2]=z;
        }
        else
        {
            maxVertices+=500;
            double[] modelTemp=new double[maxVertices*3];
            System.arraycopy(model,0,modelTemp,0,vertices*3);

            modelTemp[vertices*3]=x;
            modelTemp[vertices*3+1]=y;
            modelTemp[vertices*3+2]=z;

            model=modelTemp;

            transModel=new double[maxVertices*3];
        }
        vertices++;
    }


    void addThing(double cX,double cY,double cZ,double size)
    {
        int[] polygon=new int[4];
        int vertOffset=vertices;
```

```
    int xSteps=40;
    int ySteps=40;
    boolean visible;
    for (int y=0;y<ySteps;y++)
        for (int x=0;x<xSteps;x++)
        {
            double xd=size*2*x/xSteps-size;
            double yd=size*2*y/ySteps-size;
            double zd=0;
            addVertex(cX+xd,cY+yd,cZ+zd);
        }

    for (int y=0;y<ySteps-1;y++)
        for (int x=0;x<xSteps-1;x++)
        {
            polygon[3]=vertOffset+x+y*xSteps;
            polygon[2]=vertOffset+x+1+y*xSteps;
            polygon[1]=vertOffset+x+1+(y+1)*xSteps;
            polygon[0]=vertOffset+x+(y+1)*xSteps;
           addPolygon(polygon,Color.yellow);
       }
}

void addCube(double cX,double cY,double cZ,double size,Color c)
{
    int[] polygon=new int[4];
    int vertOffset=vertices;

    //Set up vertices to form cube
    addVertex(cX-size,cY-size,cZ-size);
    addVertex(cX-size,cY+size,cZ-size);
    addVertex(cX+size,cY+size,cZ-size);
    addVertex(cX+size,cY-size,cZ-size);

    addVertex(cX-size,cY-size,cZ+size);
    addVertex(cX-size,cY+size,cZ+size);
    addVertex(cX+size,cY+size,cZ+size);
    addVertex(cX+size,cY-size,cZ+size);

    //Set up polygons to form cube
    polygon[0]=vertOffset+0;
    polygon[1]=vertOffset+1;
    polygon[2]=vertOffset+2;
    polygon[3]=vertOffset+3;
    addPolygon(polygon,c);
```

```
 polygon[0]=vertOffset+7;
 polygon[1]=vertOffset+6;
 polygon[2]=vertOffset+5;
 polygon[3]=vertOffset+4;
 addPolygon(polygon,c);

 polygon[0]=vertOffset+4;
 polygon[1]=vertOffset+5;
 polygon[2]=vertOffset+1;
 polygon[3]=vertOffset+0;
 addPolygon(polygon,c);

 polygon[0]=vertOffset+2;
 polygon[1]=vertOffset+6;
 polygon[2]=vertOffset+7;
 polygon[3]=vertOffset+3;
 addPolygon(polygon,c);

 polygon[0]=vertOffset+1;
 polygon[1]=vertOffset+5;
 polygon[2]=vertOffset+6;
 polygon[3]=vertOffset+2;
 addPolygon(polygon,c);

 polygon[0]=vertOffset+3;
 polygon[1]=vertOffset+7;
 polygon[2]=vertOffset+4;
 polygon[3]=vertOffset+0;
 addPolygon(polygon,c);

//Update fastLines array
  int[] fastLineTemp=new int[fastLines+12*2];
  if (fastLine!=null) System.arraycopy(fastLine,0,
      fastLineTemp,0,fastLines);
fastLine=fastLineTemp;

fastLine[fastLines++]=vertOffset+0;
fastLine[fastLines++]=vertOffset+1;
fastLine[fastLines++]=vertOffset+1;
fastLine[fastLines++]=vertOffset+2;
fastLine[fastLines++]=vertOffset+2;
fastLine[fastLines++]=vertOffset+3;
fastLine[fastLines++]=vertOffset+3;
fastLine[fastLines++]=vertOffset+0;
```

```
    fastLine[fastLines++]=vertOffset+4;
    fastLine[fastLines++]=vertOffset+5;
    fastLine[fastLines++]=vertOffset+5;
    fastLine[fastLines++]=vertOffset+6;
    fastLine[fastLines++]=vertOffset+6;
    fastLine[fastLines++]=vertOffset+7;
    fastLine[fastLines++]=vertOffset+7;
    fastLine[fastLines++]=vertOffset+4;

    fastLine[fastLines++]=vertOffset+1;
    fastLine[fastLines++]=vertOffset+5;
    fastLine[fastLines++]=vertOffset+2;
    fastLine[fastLines++]=vertOffset+6;
    fastLine[fastLines++]=vertOffset+3;
    fastLine[fastLines++]=vertOffset+7;
    fastLine[fastLines++]=vertOffset+0;
    fastLine[fastLines++]=vertOffset+4;
}


void createNormals(int sign)
{
     normals=0; maxNormals=500;
      normal=new double[maxNormals*6];
      transNormal=new double[maxNormals*6];

    int[] p=new int[500]; //Max vertices in an object
    int nVertices=0;

     double midX=0,midY=0,midZ=0;
      for (int n=0;n<polyVertices;n++)
      {
          int v=poly[n];
           if (v<0)
           {
            //Find the midpoint of the polygon
            midX/=nVertices;
            midY/=nVertices;
            midZ/=nVertices;

            //Find normal
            double vectorX1=model[p[0]*3];
            double vectorY1=model[p[0]*3+1];
            double vectorZ1=model[p[0]*3+2];

            double vectorX2=model[p[1]*3];
```

```
            double vectorY2=model[p[1]*3+1];
            double vectorZ2=model[p[1]*3+2];

            double vectorX3=model[p[2]*3];
            double vectorY3=model[p[2]*3+1];
            double vectorZ3=model[p[2]*3+2];

            double px=vectorX1-vectorX2;
            double py=vectorY1-vectorY2;
            double pz=vectorZ1-vectorZ2;

            double qx=vectorX1-vectorX3;
            double qy=vectorY1-vectorY3;
            double qz=vectorZ1-vectorZ3;

            double nx=py*qz-pz*qy;
            double ny=pz*qx-px*qz;
            double nz=px*qy-py*qx;

            double length=Math.sqrt(nx*nx+ny*ny+nz*nz);
            double scale=1.0/length;

            nx*=sign*scale;
            ny*=sign*scale;
            nz*=sign*scale;

            addNormal(midX,midY,midZ,midX+nx,midY+ny,midZ+nz);

                nVertices=0; midX=0; midY=0; midZ=0;
           }
           else
           {
                p[nVertices++]=v;
            midX+=model[v*3];
            midY+=model[v*3+1];
            midZ+=model[v*3+2];
           }
      }
}

 void addPolygon(int[] p,Color c)
 {
     //The polygon is added and a normal vector is
     //constructed to point according to the left thumb rule
     int nVertices=p.length;
     double midX=0,midY=0,midZ=0;
```

```
 for (int n=0;n<nVertices;n++)
 {
     addPolyVert(p[n]);
     midX+=model[p[n]*3];
     midY+=model[p[n]*3+1];
     midZ+=model[p[n]*3+2];
 }
addPolyVert(c.getRGB());

//Find the midpoint of the polygon
midX/=nVertices;
midY/=nVertices;
midZ/=nVertices;

//Find normal
double vectorX1=model[p[0]*3];
double vectorY1=model[p[0]*3+1];
double vectorZ1=model[p[0]*3+2];

double vectorX2=model[p[1]*3];
double vectorY2=model[p[1]*3+1];
double vectorZ2=model[p[1]*3+2];

double vectorX3=model[p[2]*3];
double vectorY3=model[p[2]*3+1];
double vectorZ3=model[p[2]*3+2];

double px=vectorX1-vectorX2;
double py=vectorY1-vectorY2;
double pz=vectorZ1-vectorZ2;

double qx=vectorX1-vectorX3;
double qy=vectorY1-vectorY3;
double qz=vectorZ1-vectorZ3;

double nx=py*qz-pz*qy;
double ny=pz*qx-px*qz;
double nz=px*qy-py*qx;

double length=Math.sqrt(nx*nx+ny*ny+nz*nz);
double scale=1.0/length;

nx*=scale;
ny*=scale;
nz*=scale;
```

```
        addNormal(midX,midY,midZ,midX+nx,midY+ny,midZ+nz);

        polygons++;
    }


    void addNormal(double x1, double y1, double z1,double x2,
                           double y2, double z2)
    {
         if (normals<maxNormals)
         {
              normal[normals*6]=x1;
              normal[normals*6+1]=y1;
              normal[normals*6+2]=z1;

              normal[normals*6+3]=x2;
              normal[normals*6+4]=y2;
              normal[normals*6+5]=z2;
         }
         else
         {
              maxNormals+=500;
              double[] normalTemp=new double[maxNormals*6];
              System.arraycopy(normal,0,normalTemp,0,normals*6);

              normalTemp[normals*6]=x1;
              normalTemp[normals*6+1]=y1;
              normalTemp[normals*6+2]=z1;

              normalTemp[normals*6+3]=x2;
              normalTemp[normals*6+4]=y2;
              normalTemp[normals*6+5]=z2;

              normal=normalTemp;
              transNormal=new double[maxNormals*6];
         }
         normals++;
    }


    void addPolyVert(int p)
    {
         if (polyVertices<maxPolyVertices)
              poly[polyVertices]=p;
         else
```

```
     {
          maxPolyVertices+=500;
          int[] polyTemp=new int[maxPolyVertices];
          System.arraycopy(poly,0,polyTemp,0,polyVertices);

          polyTemp[polyVertices]=p;
          poly=polyTemp;
     }
     polyVertices++;
}

void waitForImage(Image picture)
{
     //Make sure the image is fully loaded
     MediaTracker tracker = new MediaTracker(this);
     tracker.addImage(picture, 0);
     try {
          tracker.waitForID(0);
     } catch (InterruptedException e);
}


void imageTo3D(Image picture)
{
       //First make sure the image is fully loaded
       waitForImage(picture);

     //Get the width and height of the image
     int w=picture.getWidth(this);
     int h=picture.getHeight(this);

     //Allocate buffer to hold the image's pixels
     int[] pixels = new int[w * h];

     //Grab pixels
     PixelGrabber pg = new PixelGrabber(picture, 0, 0,
            w, h, pixels, 0, w);
     try {
          pg.grabPixels();
     } catch (InterruptedException e);


   //Color to be considered the background color is
   //located in the top left corner.
   int bg=pixels[0];
   int cubeSize=200;
```

```
    for (int y=0;y<h;y++)
        for (int x=0;x<w;x++)
        {

            int col=pixels[y*w+x];
            if (bg!=col)
            {
                //Add cube to scene if the pixel is not transparent
                int r=(col&0xff0000)>>16;
                int g=(col&0xff00)>>8;
                int b=col&0xff;
                Color c=new Color(r,g,b);

                 addCube((x-w/2)*cubeSize,(h/2-y)*cubeSize,
                              0,cubeSize/2.2,c);
             }
        }
}


void imageTo3DMap(Image picture,double height)
{
       //First make sure the image is fully loaded
       waitForImage(picture);

     //Get the width and height of the image
     int w=picture.getWidth(this);
     int h=picture.getHeight(this);

     //Allocate buffer to hold the image's pixels
     int[] pixels = new int[w * h];

     //Grab pixels
     PixelGrabber pg = new PixelGrabber(picture,
         0, 0, w, h, pixels, 0, w);
     try {
          pg.grabPixels();
     } catch (InterruptedException e);


   //Set up all vertices
   double scale=(w+h)/2;
   int offset=vertices;
   for (int y=0;y<h;y++)
       for (int x=0;x<w;x++)
       {
```

```
            int col=pixels[y*w+x];
            int r=(col&0xff0000)>>16;
            int g=(col&0xff00)>>8;
            int b=col&0xff;
            double gray=(r+g+b)/3.0;
            addVertex ((x-w/2),(h/2-y),-gray*height*scale/1000);
        }

    //Set up polygons
    int[] p=new int[4];
    for (int y=0;y<h-1;y++)
        for (int x=0;x<w-1;x++)
        {
            p[0]=offset+x+y*w;
            p[1]=offset+(x+1)+y*w;
            p[2]=offset+(x+1)+(y+1)*w;
            p[3]=offset+x+(y+1)*w;
            int col=pixels[y*w+x];
            addPolygon (p,new Color(col));
        }
}


void setLight(double x,double y,double z)
{
     light[0]=x;
     light[1]=y;
     light[2]=z;
}

void moveLight(double dx,double dy,double dz)
{
     light[0]+=dx;
     light[1]+=dy;
     light[2]+=dz;
}

void setCamera(double x,double y,double z)
{
     cameraX=x;
     cameraY=y;
     cameraZ=z;
}

void moveCamera(double dx,double dy,double dz)
{
```

```
        cameraX+=dx;
        cameraY+=dy;
        cameraZ+=dz;
    }

    void rotateXYZ(double rotX, double rotY, double rotZ)
    {
        mat.unit();
        mat.xrot(rotX);
        mat.yrot(rotY);
        mat.zrot(rotZ);
        mat.translate(-cameraX,-cameraY,-(cameraZ-frontPlane));
        mat.transform(model, transModel, vertices);
        mat.transform(normal, transNormal, normals*2);
        mat.transform(light, transLight, 1);
    }

    void rotateX(double rotX)
    {
        mat.unit();
        mat.xrot(rotX);
        mat.translate(-cameraX,-cameraY,-(cameraZ-frontPlane));
    }

    void rotateY(double rotY)
    {
        mat.unit();
        mat.yrot(rotY);
        mat.translate(-cameraX,-cameraY,-(cameraZ-frontPlane));
    }

    void rotateZ(double rotZ)
    {
        mat.unit();
        mat.zrot(rotZ);
        mat.translate(-cameraX,-cameraY,-(cameraZ-frontPlane));
    }


    void renderShadow(double distance, Color c)
    {
       //Translate z coordinate of background
       distance+=-cameraZ+frontPlane;
        GC.setColor(c);
```

```
double lightX;
double lightY;
double lightZ;
if (!cameraLightOn)
{
    lightX=transLight[0];
    lightY=transLight[1];
    lightZ=transLight[2];
}
else
{
    lightX=light[0];
    lightY=light[1];
    lightZ=light[2]-frontPlane;
}


  //Project vertices
  int[] projX=new int[vertices];
  int[] projY=new int[vertices];
  int i=0;
double s=frontPlane/distance;
  for (int n=0;n<vertices*3;n+=3)
  {

    //Find the intersection point between the
    //background plane and the line defined by
    //the vertex and the light coordinates
    double shadowX=((transModel[n]-lightX)*(distance-
         lightZ))/(transModel[n+2]-lightZ)+lightX;
    double shadowY=((transModel[n+1]-lightY)*(distance-
         lightZ))/(transModel[n+2]-lightZ)+lightY;

    //Store the projected shadow vertices
    projX[i]=(int)(shadowX*s)+centerX;
    projY[i]=(int)(-shadowY*s)+centerY;

       i++;
  }

//Draw shadow
  Polygon p=new Polygon();
  int v0=poly[0];
  for (int n=0;n<polyVertices;n++)
  {
       int v=poly[n];
```

```
            if (v<0)
            {
                 p.addPoint(projX[v0],projY[v0]);
                 v0=poly[n+1];
                 GC.fillPolygon(p);
                 p=new Polygon();
            }
            else
                 p.addPoint(projX[v],projY[v]);
       }
  }


   void render(int renderMode)
   {

      double lightX;
      double lightY;
      double lightZ;
      if (!cameraLightOn)
      {
          lightX=transLight[0];
          lightY=transLight[1];
          lightZ=transLight[2];
      }
      else
      {
          lightX=light[0];
          lightY=light[1];
          lightZ=light[2]-frontPlane;
      }

        switch(renderMode)
        {
             case 0: //Plain perspective projection of vertices only
                  //Reset z-buffer and screen-buffer
                  System.arraycopy(zBufferReset,0,zBuffer,0,
                         width*height);
                  System.arraycopy(screenBufferReset,0,
                         screenBuffer,0,width*height);
                  double colorStep=256.0/(backPlane-frontPlane);
                  for (int n=0;n<vertices*3;n+=3)
                  {
                       int z=(int)(transModel[n+2]);
                       if (z>=frontPlane)
```

```
                {
                    int x=(int)(transModel[n]);
                    int y=(int)(transModel[n+1]);
                    int xProj=x*frontPlane/z+centerX;
                    int yProj=-y*frontPlane/z+centerY;
                    if ((xProj>=0)&&(xProj<width)&&
                        (yProj>=0)&&(yProj<height))
                    {
                        int off=xProj+yProj*width;
                        if (z<zBuffer[off])
                        {
                            screenBuffer[off]=colorGrades2[
                             (int)((z-
                            frontPlane)*colorStep)];
                            zBuffer[off]=z;
                        }
                    }
                }
            }
            Image screen= createImage(new MemoryImageSource(
                width, height, screenBuffer, 0, width));
            GC.drawImage(screen,0,0,this);
        break;
        case 1: //Plain wireframe perspective projection without
                //depth shading
               //Very fast, but no z-clipping!
            GC.setColor(color);

            //Project vertices
            int[] projX=new int[vertices];
            int[] projY=new int[vertices];
            int i=0;
            for (int n=0;n<vertices*3;n+=3)
            {
                int x=(int)(transModel[n]);
                int y=(int)(transModel[n+1]);
                int z=(int)(transModel[n+2]);
                projX[i]=x*frontPlane/z+centerX;
                projY[i]=-y*frontPlane/z+centerY;
                i++;
            }

            Polygon p=new Polygon();
            int v0=poly[0];
            for (int n=0;n<polyVertices;n++)
            {
```

```
                int v=poly[n];
                if (v<0)
                {
                     p.addPoint(projX[v0],projY[v0]);
                     v0=poly[n+1];
                     GC.drawPolygon(p);
                     p=new Polygon();
                }
                else
                     p.addPoint(projX[v],projY[v]);
           }

     break;
     case 2:
           //Wireframe perspective projection with depth
           //shading and z-buffering
           //Reset z-buffer and screen-buffer
           System.arraycopy(zBufferReset,0,zBuffer,0,
                width*height);
           System.arraycopy(screenBufferReset,0,
                screenBuffer,0,width*height);
           colorStep=256.0/(backPlane-frontPlane);
           //Project vertices
           projX=new int[vertices];
           projY=new int[vertices];
           i=0;
           for (int n=0;n<vertices*3;n+=3)
           {
                int x=(int)(transModel[n]);
                int y=(int)(transModel[n+1]);
                int z=(int)(transModel[n+2]);
                projX[i]=x*frontPlane/z+centerX;
                projY[i]=-y*frontPlane/z+centerY;
                i++;
           }

           colorStep=256.0/(backPlane-frontPlane);
           for (int n=0;n<fastLines;n+=2)
           {
                int v1=fastLine[n];
                int v2=fastLine[n+1];
                int x1=projX[v1];
                int y1=projY[v1];
                int x2=projX[v2];
                int y2=projY[v2];
```

```
double z1=transModel[v1*3+2];
double z2=transModel[v2*3+2];

//Draw line using fast integer differential
//algorithm
int xInc,yInc;
int dx=x2-x1;
int dy=y2-y1;
if (Math.abs(dx)>Math.abs(dy))
{
     if (dx<0)
     {
          //Swap x1 and x2, y1 and y2
          int q=x1; x1=x2; x2=q;
          q=y1; y1=y2; y2=q;
          double q2=z1; z1=z2; z2=q2;
     }
     if (y2>y1)
          yInc=1;
     else
          yInc=-1;
     //First octant algorithm
     dx=x2-x1; dy=Math.abs(y2-y1);
     int x=x1, y=y1, error=-(dx/2);
     double z=z1,dz=(z2-z1)/dx;
     x2=Math.min(width-1,x2);
     int sOff=x+y*width;
     if ((x>=0)&&(x<width)&&(y>=0)&&
          (y<height)&&(z>=frontPlane))
     {
          if ((zBuffer[sOff]>(int)z)&&(z>0))
          {
               screenBuffer[sOff]=colorGrades2[
                (int)((z-
                   frontPlane)*colorStep)];
               zBuffer[sOff]=(int)z;
          }
     }
     while (x<x2)
     {
          error+=dy;
          if (error>=0)
          {
               y+=yInc;
               error-=dx;
          }
```

```
                x++;
                z+=dz;
                sOff=x+y*width;
                if ((x>=0)&&(y>=0)&&(y<
                     height)&&(z>=frontPlane))
                {
                     if (zBuffer[sOff]>(int)z)
                     {
                          screenBuffer[sOff]=
                             colorGrades2[
                             (int)((z-frontPlane)*
                             colorStep)];
                          zBuffer[sOff]=(int)z;
                     }
                }
           }
      }
      else
      {
           if (dy<0)
           {
                //Swap x1 and x2, y1 and y2
                int q=x1; x1=x2; x2=q;
                q=y1; y1=y2; y2=q;
                double q2=z1; z1=z2; z2=q2;
           }
           if (x2>x1)
                xInc=1;
           else
                xInc=-1;
           //Second octant algorithm
           dx=Math.abs(x2-x1); dy=y2-y1;
           int x=x1, y=y1, error=-(dy/2);
           double z=z1,dz=(z2-z1)/dy;
           y2=Math.min(height-1,y2);
           int sOff=x+y*width;
           if ((x>=0)&&(x<width)&&(y>=0)&&
                (y<height)&&(z>=frontPlane))
           {
                if (zBuffer[sOff]>(int)z)
                {
                     screenBuffer[sOff]=colorGrades2[
                     (int)((z-frontPlane)*colorStep)];
                     zBuffer[sOff]=(int)z;
                }
```

```
                    }
                    while (y<y2)
                    {
                         error+=dx;
                         if (error>=0)
                         {
                              x+=xInc;
                              error-=dy;
                         }
                         y++;
                         z+=dz;
                         sOff=x+y*width;
                         if ((x>=0)&&(x<width)&&
                             (y>=0)&&(z>=frontPlane))
                         {
                              if (zBuffer[sOff]>(int)z)
                              {
                                   screenBuffer[sOff]=
                                      colorGrades2[(int)
                                      ((z-frontPlane)*
                                      colorStep)];
                                   zBuffer[sOff]=(int)z;
                              }
                         }
                    }
               }
          }
          screen=createImage(new MemoryImageSource(
              width, height, screenBuffer, 0, width));
          GC.drawImage(screen,0,0,this);
     break;
     case 3:
     case 4:
     case 5:
          //Solid rendering with one light
          //source and hidden surface
          //elimination.
          //Mode 4 has colored surfaces, too
          //Mode 5 is mode four with no normal
          //vector surface elimination

          int polyIndex=0;
          //Project vertices
          projX=new int[vertices];
          projY=new int[vertices];
```

```
        i=0;

        for (int n=0;n<vertices*3;n+=3)
        {
             int x=(int)(transModel[n]);
             int y=(int)(transModel[n+1]);
             int z=(int)(transModel[n+2]);
             projX[i]=x*frontPlane/z+centerX;
             projY[i]=-y*frontPlane/z+centerY;
             i++;
        }

    long[] sortArray=new long[polygons];
    int visiblePolygons=0;
    int polygonOffset=0;

        colorStep=256.0/(backPlane-frontPlane);
        p=new Polygon();
        v0=poly[0];

        for (int n=0;n<polyVertices;n++)
        {
             int v=poly[n];
             if (v<0)
             {
                 //Polygon closed
                  p.addPoint(projX[v0],projY[v0]);
                  v0=poly[n+1];
                  int offset=polyIndex*6;
                 double dx=transNormal[offset+3]-
                     transNormal[offset];
             double dy=transNormal[offset+3+1]-
                     transNormal[offset+1];
1                        double dz=transNormal[offset+3+2]-
                     transNormal[offset+2];
             if (((transNormal[offset]*dx+transNormal
                 [offset+1]*dy+transNormal[offset+2]*
                 dz)<=0)||(renderMode==5))
             {
                 //The face of the polygon is headed toward
                 //the viewer, so it's visible.
                 //Or mode 5 is used

                 //Calculate the amount of light the polygon
                 //receives from the light source
                 double normalx=transNormal[offset+3]-
                    transNormal[offset];
```

```
                double normaly=transNormal[offset+3+1]-
                   transNormal[offset+1];
                double normalz=transNormal[offset+3+2]-
                   transNormal[offset+2];

                dx=lightX-transNormal[offset];
                dy=lightY-transNormal[offset+1];
                dz=lightZ-transNormal[offset+2];
                double illumination=(dx*normalx+dy*
                   normaly+dz*normalz)/Math.sqrt(dx*
                   dx+dy*dy+dz*dz);

               double xMid=transNormal[offset];
               double yMid=transNormal[offset+1];
               double zMid=transNormal[offset+2];
               int c=(int)(128*illumination)+128;

                //Store the index and the color of
                //this polygon in the distance-
                //from-camera array to be sorted
                //afterward
                //This value could be replaced with
                //zMid for slightly faster
                //computing, but it's a tad more
                //inaccurate
                long d=(long)(Math.sqrt(xMid*xMid+
                    yMid*yMid+zMid*zMid));

               //Depth shading; simply remove if
               //unwanted
                c-=(int)((d-frontPlane)*colorStep);
                     if (c<0) c=0;

                sortArray[visiblePolygons++]=
                    (d<<32)+(c<<16)+polygonOffset;
                }
                 polyIndex++;
                 p=new Polygon();
                 polygonOffset=n+1;
           }
           else
                p.addPoint(projX[v],projY[v]);
      }
```

```
  //Do the sorting
  //If you have only one convex object
  //You can drop this, the hidden surface
  //algorithm using normal vectors will do the
  //job for you
  sort(sortArray, 0, visiblePolygons-1);

if (renderMode==3)
{
        //Draw monochrome polygons in sorted
        //order
    for (int n=visiblePolygons-1;n>=0;n--)
    {
        long l=sortArray[n];
        //Extract light intensity (0-255)
            int c=(int)((l>>16)&0xff);
            //Extract polygon offset
            i=(int)(l&0xffff);
             p=new Polygon();
             int v=poly[i];
        do
        {
                  p.addPoint(projX[v],projY[v]);
            v=poly[++i];
        } while (v>=0);

            GC.setColor(colorGrades[c]);
             GC.fillPolygon(p);
    }
}
else
{
        //Draw colored polygons in sorted order
    for (int n=visiblePolygons-1;n>=0;n--)
    {
        long l=sortArray[n];
        //Extract light intensity (0-255)
            int c=(int)((l>>16)&0xff);
            //Extract polygon offset
            i=(int)(l&0xffff);
             p=new Polygon();
             int v=poly[i];
        do
        {
                  p.addPoint(projX[v],projY[v]);
            v=poly[++i];
```

```
                    } while (v>=0);

                    int r=(((v&0xff0000)*c)>>8)&0xff0000;
                    int g=(((v&0xff00)*c)>>8)&0xff00;
                    int b=((v&0xff)*c)>>8;

                        GC.setColor(new Color(r+g+b));
                         GC.fillPolygon(p);
                }
             }

            break;
        }
}

//Implementation of the QuickSort algorithm based on SUNs
//more general class library
void sort(long a[], int lo0, int hi0)
{
     int lo = lo0;
     int hi = hi0;
     if ((hi-lo) <= 3)
     {
          brute(a, lo, hi);
          return;
     }
     long mid = a[(lo + hi) / 2];
     while (lo < hi)
     {
          while (lo<hi && a[lo] < mid)
          lo++;

          while (lo<hi && a[hi] >= mid)
          hi--;

          if (lo < hi)
          {
               long T = a[lo];
               a[lo] = a[hi];
               a[hi] = T;
          }
     }
     if (hi < lo)
     {
          int T = hi;
```

```
            hi = lo;
            lo = T;
        }
        sort(a, lo0, lo);
        sort(a, lo == lo0 ? lo+1 : lo, hi0);
    }

    void brute(long a[], int lo, int hi)
    {
        if ((hi-lo) == 1)
        {
            if (a[hi] < a[lo])
            {
                long T = a[lo];
                a[lo] = a[hi];
                a[hi] = T;
            }
        }
        if ((hi-lo) == 2)
        {
            int pmin = a[lo] < a[lo+1] ? lo : lo+1;
            pmin = a[pmin] < a[lo+2] ? pmin : lo+2;
            if (pmin != lo)
            {
                long T = a[lo];
                a[lo] = a[pmin];
                a[pmin] = T;
            }
            brute(a, lo+1, hi);
        }
        if ((hi-lo) == 3)
        {
            int pmin = a[lo] < a[lo+1] ? lo : lo+1;
            pmin = a[pmin] < a[lo+2] ? pmin : lo+2;
            pmin = a[pmin] < a[lo+3] ? pmin : lo+3;
            if (pmin != lo)
            {
                long T = a[lo];
                a[lo] = a[pmin];
                a[pmin] = T;
            }
            int pmax = a[hi] > a[hi-1] ? hi : hi-1;
            pmax = a[pmax] > a[hi-2] ? pmax : hi-2;
            if (pmax != hi)
            {
```

```
                    long T = a[hi];
                    a[hi] = a[pmax];
                    a[pmax] = T;
                }
                brute(a, lo+1, hi-1);
            }
        }
}
```

Wrapping Up

By now you should know your way around the world of 3D computer graphics. This is a vast field, and although the 3D libraries presented in this book are comprehensive, they really only begin to touch on the subject. But it's a good place to start, and all the dirty work has already been done for you. You can do a lot of fancy things by spicing up the texture-mapping routines some, such as adding routines for mapping textures on cubes, spheres, and other shapes. You should also experiment some more with multiple light sources, which should be fairly easy to implement once you get familiar with this class. The possibilities are endless.